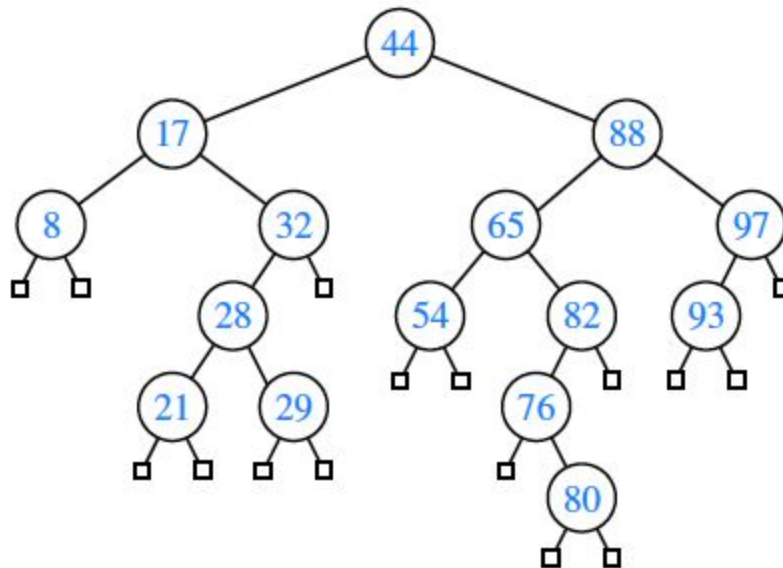


Binary Search Tree

BST:

- A **binary search tree(BST)** is a tree where the key in any node is larger than the keys in all nodes in the node's left subtree and smaller than the keys in all nodes in that nodes right subtree
- This property allows us to efficiently search for a key
- Each subtree is itself a binary search tree
- The left and right pointers for leaf nodes are sentinels that are used as placeholders. But we can cut down on the memory used by carefully using null references instead.
- In order traversal of a BST will go through the elements in sorted order.



Source: Goodrich

Searching:

- We begin by checking the root's key with our target key.
 - If it is the key we are looking for then return true, we found the target key.
 - If target's key < root's key, search the left subtree.
 - If target's key > root's key, search the right subtree.
- We repeat this process until we find the target (successful search) or reach a null subtree (unsuccessful search).

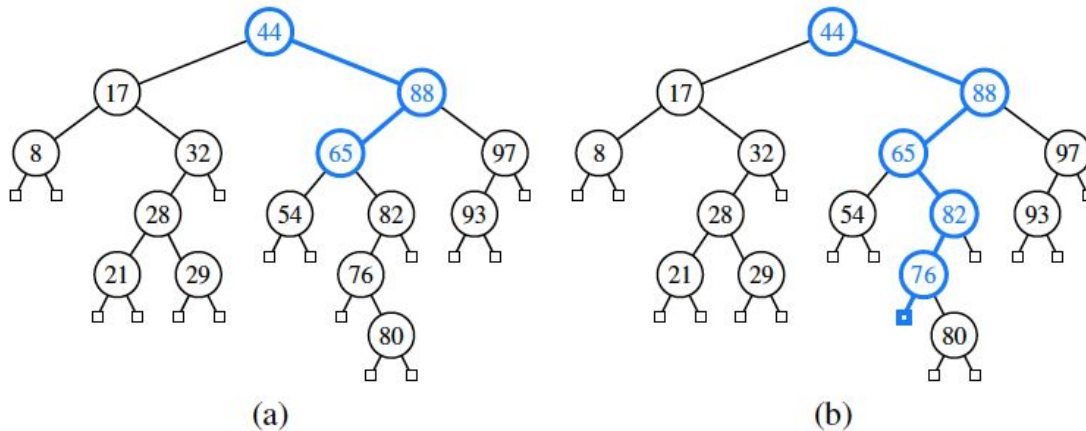


Figure 11.2: (a) A successful search for key 65 in a binary search tree; (b) an unsuccessful search for key 68 that terminates at the leaf to the left of the key 76.

Source: Goodrich

Analysis:

- The running time is dependent on the shape of the tree which in turn depends on the order in which the keys are inserted.
- We assume that the keys are random or that they are inserted in a random order. Then with each recursive call, the node moves down a level in the tree. Starting from the root until it hits a leaf node.
- At best, this number is bounded by $h + 1$, where h is the height of the tree.
- If the tree is perfectly balanced, then the runtime is $O(h)$ or $O(\lg n)$
- But if the tree is not balanced, then at worst the nodes were inserted in sorted order.
 - Then the tree height will be n , and the runtime will be $O(n)$
- The balance in typical trees turns out to be much closer to the best case than worst case.

Inserting:

- Inserting is similar to searching
- First we search for the key we want to insert.
 - If it exists then do nothing as we already have this key
 - If it doesn't, then the null reference we hit on the path to check if the key already exist, replace that null reference with a new node containing the new key.

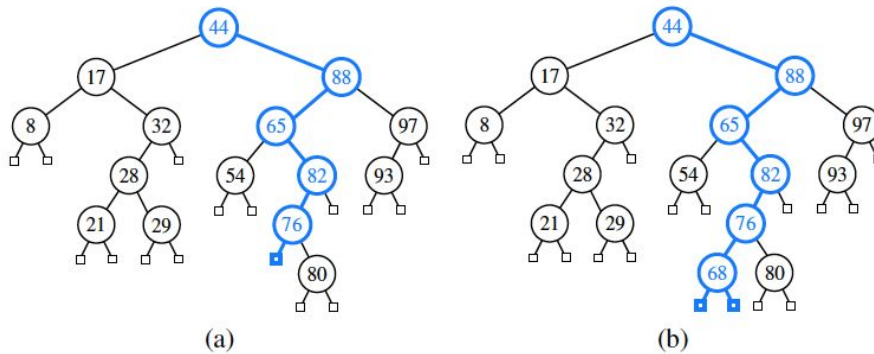


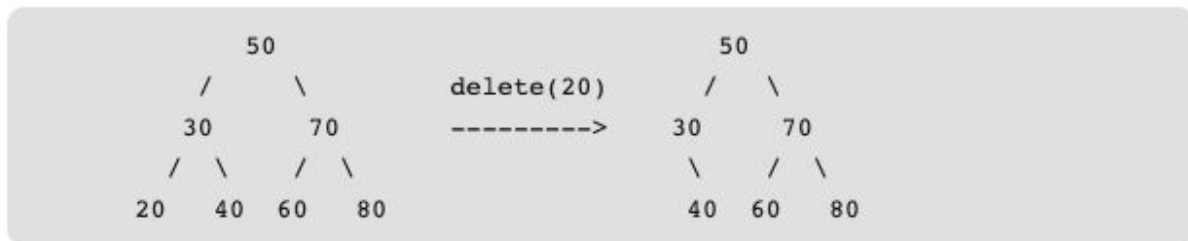
Figure 11.4: Insertion of an entry with key 68 into the search tree of Figure 11.2. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

Source: Goodrich

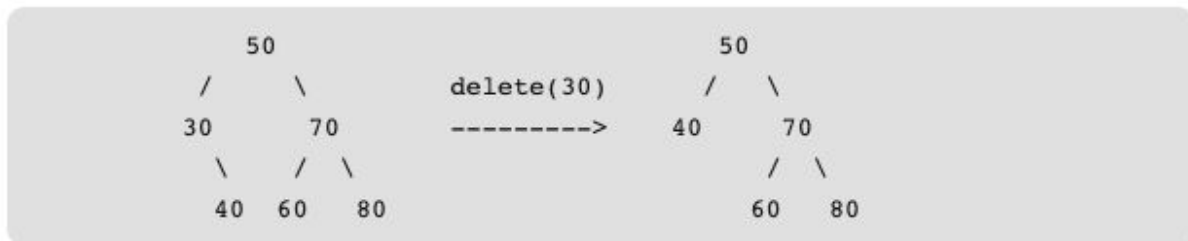
Deleting:

- First we find the node we want to remove, call this node *targetNode*
- There are 3 possible situations can occur:
 - No children
 - Update parent's left or right pointer to null (depending on whether *targetNode* is a left child or a right child).
 - 1 Child
 - Update parent's pointer to point to *targetNode*'s child.
 - 2 Children
 - Replace *targetNode* with its **successor** (the successor is the smallest node in the right subtree or leftmost node).
 - Then delete the successor from the right subtree (deleting the successor will be one of the first 2 cases).

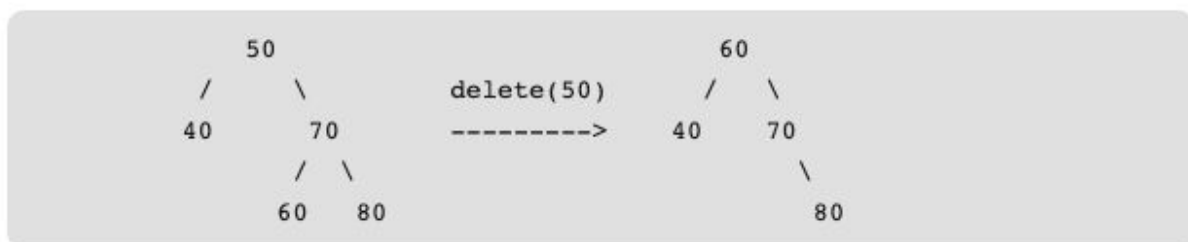
1) **Node to be deleted is leaf:** Simply remove from the tree.



2) **Node to be deleted has only one child:** Copy the child to the node and delete the child



3) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



Source: <https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/>